Introduction to python

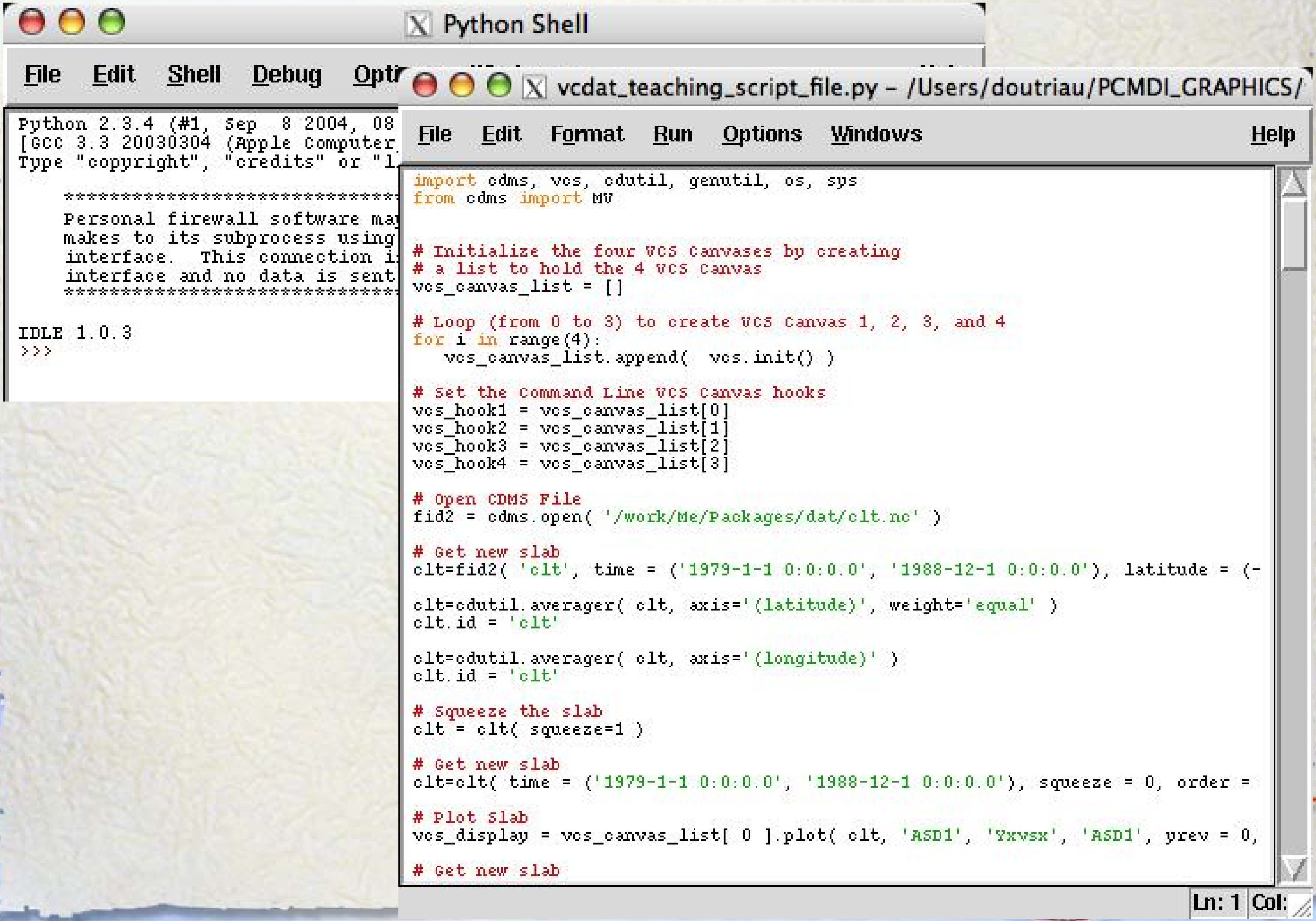
Introduction and Concepts

- Why Python ?
  - Very simple syntax with clean semantics
  - Free and used extensively
  - Interpreted, quick development and debug
  - Virtually every area of computer science
  - Easily extensible in C or C++ and other languages (FORTRAN TOO!)
  - Object Oriented (but no need to know about it)

Python Editors – Idle (1)

- IDLE is the new Python development environment which was first released with version 1.5.2 of Python
- Its name is an acronym of "Integrated DeveLopment Environment".
- It is being developed by Guido van Rossum with contributions from others
- It has a Python Shell Window, which gives you access to the Python interactive mode
- Its File Editor lets you create new or browse through and edit existing Python source files (color coded).
- There is a Path Browser for searching through the path of available module source files as well as a simple Class Browser for finding the methods of classes.
- It has a flexible search capability through its Find in Files dialog that lets you search through your files and/or the systems' files to find occurrences of identifiers or any other text fragments.
- Finally (although this is still is in the process of maturing), it has a Debug Control Panel which provides for the symbolic debugging of Python programs

Python Editors – Idle (2)


Python Shell
File Edit Shell Debug Opti
vcdat_teaching_script_file.py - /Users/doutriau/PCMDI_GRAPHICS/
File Edit Format Run Options Windows Help
Ln: 1 Col:

```
Python 2.3.4 (#1, Sep  8 2004, 08
[GCC 3.3 20030304 (Apple Computer
Type "copyright", "credits" or "l

    ********************************
    Personal firewall software may
    makes to its subprocess using
    interface.  This connection i
    interface and no data is sent
    ********************************

IDLE 1.0.3
>>>
```

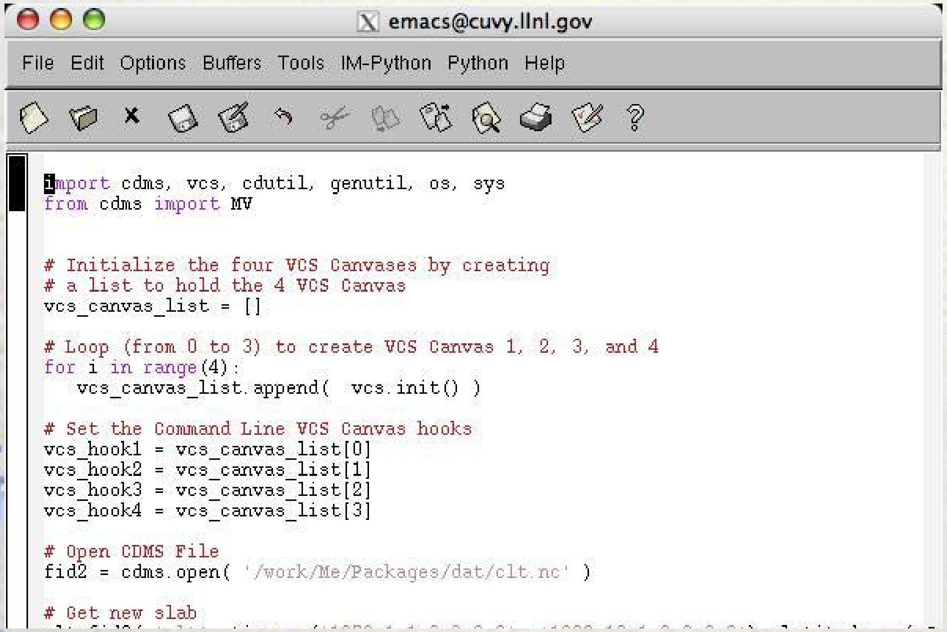
```
import cdms, vcs, cdutil, genutil, os, sys
from cdms import MV


# Initialize the four VCS Canvases by creating
# a list to hold the 4 VCS Canvas
vcs_canvas_list = []

# Loop (from 0 to 3) to create VCS Canvas 1, 2, 3, and 4
for i in range(4):
   vcs_canvas_list.append( vcs.init() )

# Set the Command Line VCS Canvas hooks
vcs_hook1 = vcs_canvas_list[0]
vcs_hook2 = vcs_canvas_list[1]
vcs_hook3 = vcs_canvas_list[2]
vcs_hook4 = vcs_canvas_list[3]

# Open CDMS File
fid2 = cdms.open( '/work/Me/Packages/dat/clt.nc' )

# Get new slab
clt=fid2( 'clt', time = ('1979-1-1 0:0:0.0', '1988-12-1 0:0:0.0'), latitude = (-

clt=cdutil.averager( clt, axis='(latitude)', weight='equal' )
clt.id = 'clt'

clt=cdutil.averager( clt, axis='(longitude)' )
clt.id = 'clt'

# Squeeze the slab
clt = clt( squeeze=1 )

# Get new slab
clt=clt( time = ('1979-1-1 0:0:0.0', '1988-12-1 0:0:0.0'), squeeze = 0, order =

# Plot Slab
vcs_display = vcs_canvas_list[ 0 ].plot( clt, 'ASD1', 'Yxvsx', 'ASD1', yrev = 0,

# Get new slab
```

Python Editors - Emacs

EMACS has a Python mode which you can obtain from http://www.python.org

I recommend to save file and run it from command line, it's faster:

```
python -i myfile.py
```



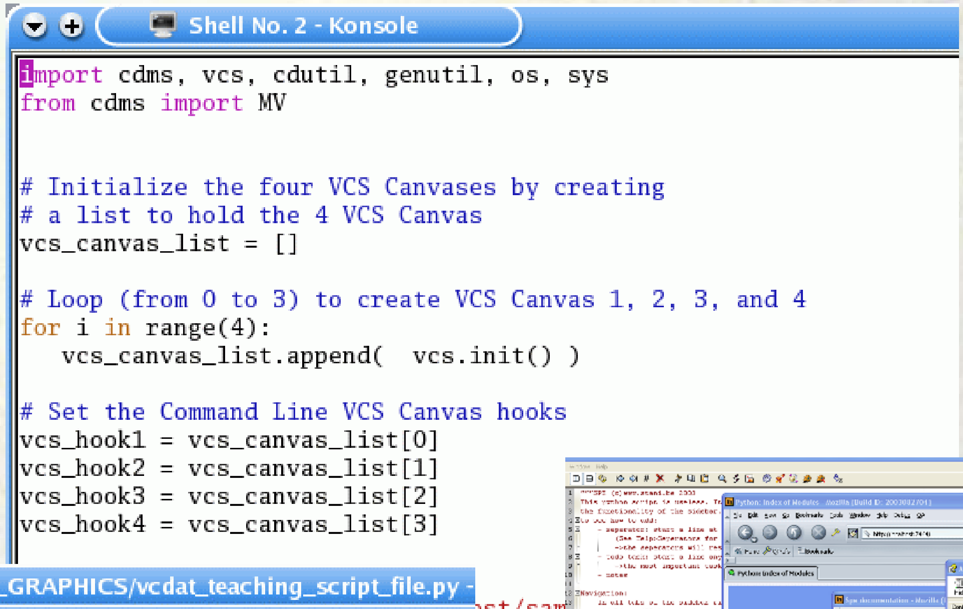
```python
import cdms, vcs, cdutil, genutil, os, sys
from cdms import MV


# Initialize the four VCS Canvases by creating
# a list to hold the 4 VCS Canvas
vcs_canvas_list = []

# Loop (from 0 to 3) to create VCS Canvas 1, 2, 3, and 4
for i in range(4):
   vcs_canvas_list.append(  vcs.init() )

# Set the Command Line VCS Canvas hooks
vcs_hook1 = vcs_canvas_list[0]
vcs_hook2 = vcs_canvas_list[1]
vcs_hook3 = vcs_canvas_list[2]
vcs_hook4 = vcs_canvas_list[3]

# Open CDMS File
fid2 = cdms.open( '/work/Me/Packages/dat/clt.nc' )

# Get new slab
```

```lisp
;; Red Hat Linux default .emacs initialization file  ; -*- mode: emacs-lisp -*-

;; Set up the keyboard so the delete key on both the regular keyboard
;; and the keypad delete the character under the cursor and to the right
;; under X, instead of the default, backspace behavior.
(global-set-key [delete] 'delete-char)
(global-set-key [kp-delete] 'delete-char)

;; turn on font-lock mode
(global-font-lock-mode t)
;; enable visual feedback on selections
(setq-default transient-mark-mode t)

;; always end a file with a newline
(setq require-final-newline t)

;; stop at the end of the file, not just add lines
(setq next-line-add-newlines nil)

(when window-system
  ;; enable wheelmouse support by default
  (mwheel-install)
  ;; use extended compound-text coding for X clipboard
  (set-selection-coding-system 'compound-text-with-extensions))

;; Charles' addition, goto (F5) and save (F6)
(global-set-key [f5] 'goto-line)
(global-set-key [f6] 'save-buffer)
(global-set-key [f1] 'advertised-undo)
(global-set-key [f2] 'query-replace-regexp)

;;; Trying to set the postscript printer.....
(setq ps-printer-name "xeroxcolor")
;;(setq ps-printer-name "hpcolor")
(setq load-path (cons "~/emacs/" load-path))
(autoload 'python-mode "python-mode" "Python editing mode." t)
(setq auto-mode-alist
        (cons '("\\.py$" . python-mode)
         auto-mode-alist))

   (setq interpreter-mode-alist
        (cons '("python" . python-mode)
         interpreter-mode-alist))
(custom-set-variables
  ;; custom-set-variables was added by Custom -- don't edit or cut/paste it!
  ;; Your init file should contain only one such instance.
 '(case-fold-search t)
 '(current-language-environment "UTF-8")
 '(default-input-method "rfc1345")
 '(global-font-lock-mode t nil (font-lock))
 '(show-paren-mode t nil (paren))
 '(transient-mark-mode t))
(custom-set-faces
  ;; custom-set-faces was added by Custom -- don't edit or cut/paste it!
  ;; Your init file should contain only one such instance.
 )
```

Editors - and many more

VI/VIM

Simple works everywhere

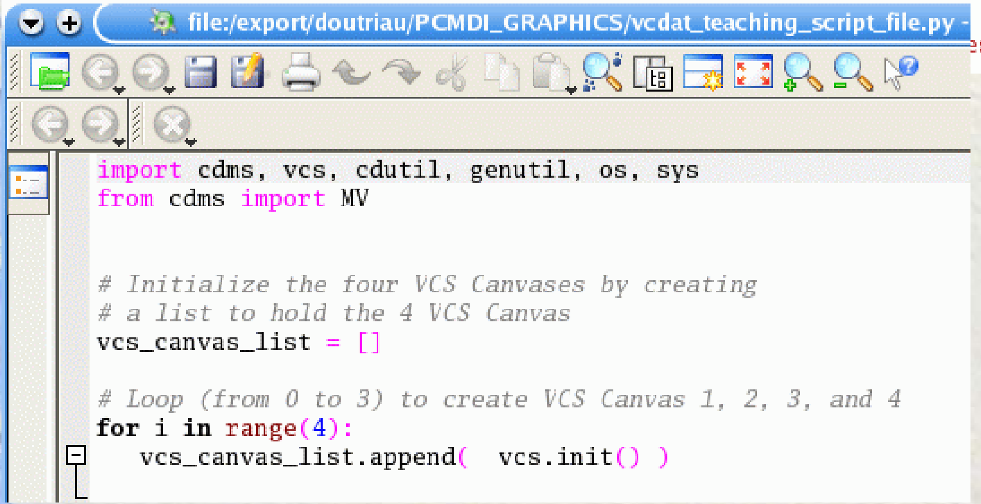


```
import cdms, vcs, cdutil, genutil, os, sys
from cdms import MV


# Initialize the four VCS Canvases by creating
# a list to hold the 4 VCS Canvas
vcs_canvas_list = []

# Loop (from 0 to 3) to create VCS Canvas 1, 2, 3, and 4
for i in range(4):
   vcs_canvas_list.append(  vcs.init() )

# Set the Command Line VCS Canvas hooks
vcs_hook1 = vcs_canvas_list[0]
vcs_hook2 = vcs_canvas_list[1]
vcs_hook3 = vcs_canvas_list[2]
vcs_hook4 = vcs_canvas_list[3]
```

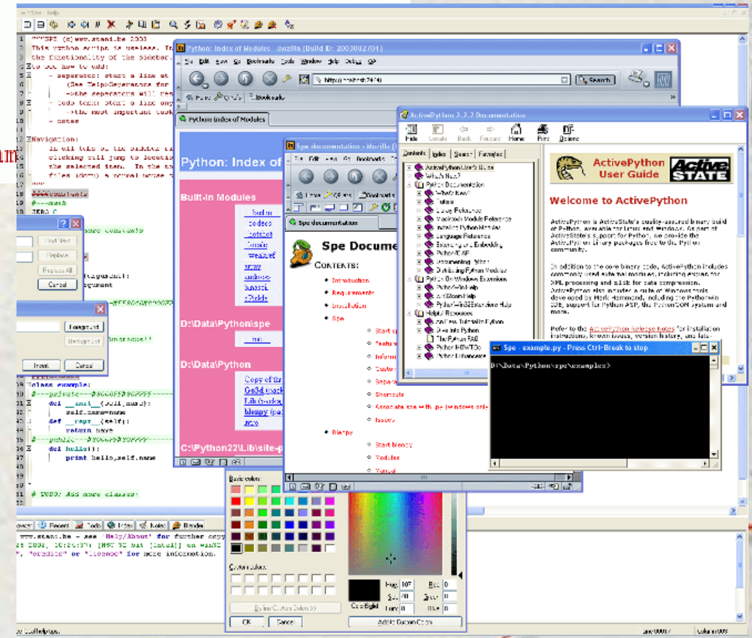
```
est/sam
```

SPE

KDevelop


file:/export/doutriau/PCMDI_GRAPHICS/vcdat_teaching_script_file.py -

```
import cdms, vcs, cdutil, genutil, os, sys
from cdms import MV


# Initialize the four VCS Canvases by creating
# a list to hold the 4 VCS Canvas
vcs_canvas_list = []

# Loop (from 0 to 3) to create VCS Canvas 1, 2, 3, and 4
for i in range(4):
   vcs_canvas_list.append(  vcs.init() )
```



Before we start: remember “dir()” and “help()”

- Whenever you are using Python interactively and you get stuck, try:
  - `dir(<something>)` OR `help(<something>)`
- For example:

```
>>> help(open)
Help on class file in module __builtin__:
class file(object)
| file(name[, mode[, buffering]]) -> file object
|
|  Open a file.  The mode can be 'r', 'w' or 'a' for
    reading (default),
```

Really remember “dir()” and “help()”

```
>>> dir("somestring")
['__add__', '__class__', '__contains__',
'__delattr__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__','__ne__',
'__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize',
'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index',
'isalnum', 'isalpha', 'isdigit','islower',
'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'replace', 'rfind',
'rindex', 'rjust', 'rstrip', 'split',
'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper',
'zfill']
```

- This will make more sense later!

Python Types

- None : represent “nothing” (NULL)
- Numbers No int or float, only long and double
- List: [ ]
  - Ordered list of elements, which can be changed
- Tuples: ( )
  - Ordered list of elements, which cannot be altered
- Dictionaries { }
  - Unordered list of keys/values
- String "", ‘ ‘ or """ """
  - Character strings
- Files
- Functions : def
  - User defined function
- Class: class
  - User defined “object” contains attributes and functions

Number Types

- Numbers are always “double precision” in python, i.e., although they are called “int” and “float” integers are always really “long” and real are always “double”.
- There’s also a complex number type:

```
A=complex(4,5)
Print A # (4+5j)
```

- Attention to casting:
  - While different casting will be cast “safely”

```
3.+5 is 8.0
3+5. is 8.0
```

  - Same “kind” casting will stay in the same kind

```
3+5 is 8
3/5 is 0
```

  - Again, different kind casting will work:

```
3/5. Is 0.59999999998
3./5 is 0.59999999998
float(3)/5 is 0.59999999998
```

Types: List 1

- List: [] , list are
  - succession of object:

```
L= [1,2,3,4,7,6]; L2=[L,2,3,4]
print  L2 # [[1, 2, 3, 4, 7, 6], 2, 3, 4]
```

  - Elements start at zero: `print L[0] # 1`
  - Elements are changeable:

```
L[0]=6 ; print L # [6,2,3,4,7,6]
```

  - Sortable: `L.sort(); print L # [2,3,4,6,6,7]`

- Accessing elements:
  - By index, negative values allowed: `L[0] is 2, L[-1] is 7, L[1] is 3 L[-4] is 4`
  - By value (I.e. returning the index of a value)

```
A=L.index(4) # returns 6
```

  - By slice: `L[2:5], L[:-2] , L[2:], L[2:-2]`

Types: List 2

- Adding elements:

```
L.append(3.5) # L is [2,3,4,6,6,7,3.5]
L.insert(2,4.5) # L is [2,3,4.5,4,6,6,7,3.5] # first
  argument is element before which to insert
```

- Removing elements

```
del(L[3]) # L is [2,3,4.5,6,6,7,3.5]
A=L.pop(-1); print A # returns 3.5 for A and L is
  [2,3,4.5,6,6,7]
```

- Other functions:

```
L.reverse() # inverse the elements order
L.count(6) # returns 2 because 6 appears twice
len(L) # returns 6 because L has 6 elements
```

Types: Tuple

- Tuples: () , tuples are
  - Ordered succession of object:

```
T= (1,2,3,4,7,6); T2=(T,2,3,4]);
print  T2 # ((1, 2, 3, 4, 7, 6), 2, 3, 4)
```

  - Elements start at zero: `print T[0] # 1`
  - Elements are NOT changeable:

```
T[0]=6  # Raises an exception
```

  - **Sortable:** `T.sort(); print T # [2,3,4,6,6,7]`

- Adding elements: *Impossible*
- Removing elements: *Impossible*

- Accessing elements:
  - By index, negative values allowed:

```
T[0] is 2 T[-1] is 7, T[1] is 3 T[-4] is 4.5
```

- Other functions:

```
len(T) # returns 6 because T has 6 elements
```

Types: Dictionary 1

- Dictionaries: {}, dictionaries are
  - Unordered pairs of key/value:
    - Keys can be any python object except list, tuple or dictionaries
    - Values can be any python object

    ```
    D={1:’one’, ‘two’:2, ‘list’:[1,2]}
    ```
  - Elements are changeable D[1]=‘One’

- Adding elements:

  ```
  D[‘three’]=3.0
  ```

- Removing elements

  ```
  del(D[1])
  A=D.poo(‘three’) # A is now 3. And D does not
     contain ‘three’ key anymore
  ```

Types: Dictionary 2

- Accessing elements:

  `print D[‘two’] # return 2`

  `A=d.get(‘three’,3.)` # Returns the value associated with ‘three’ if present, if not returns 3., second arg is optional, raise an exception if key doesn’t exist

  `A.setdefault(‘three’,3.)` # Same as above but also adds the ‘three’ key if doesn’t exist

- Other functions:

```
D.keys() # returns a list of the keys
D.values() # return a list of the contains values
D.clear() remove all pairs key/value
D.has_key() # returns 1 if key exist
```

Types: String (1)

- String are the main reason Python is so adapted to our field, it is VERY easy to manipulate here's a list of key function for strings
- Manipulating strings:

  `S=“ Welcome \n  My name is Charles. We are in a Python Class”`

  `A=‘Hi’ ; B=‘there’ ; C=A+B ;  D=A+’ ‘+B` # ‘C is ‘Hithere’, D is “Hi there”

  `I=s.find(‘Charles’)` # returns location of first occurrence of ‘Charles’ ; -1 if not present

  `S2=S.replace(‘Charles’, ‘Charles Doutriaux’)`

  `S2=S.strip()` # removes leading and ending blanks

  `L=S.split()` # returns a list of strings separation happen after every “space” or end of line (“\n”)

Types: String (2)

`L=S.split('.')` # same but splits only when finding '.' string

`L=S.split('Charles')` # same but splits when finding 'Charles' string

`S2=S.lower()` # everything becomes lower case (also `S.upper()` and `S.swapcase()`)

- Most function are accessible from the **string module** but requires then to have S passed as first arg:

```
S2=string.lower(S)
I=string.find(S,'Charles')
```

- Special Characters:
  - `'\n'` : line break
  - `'\t'` : tab
  - `'\''` : single quote when within single quote : S= ' Hi I\'m going home'
  - `"\""` : a double quote when between double quotes

Types: String : Input/Conversion

- Converting strings is easy

```
S=‘2.3’; f=float(S)
S=‘2.3’; I=int(s)
```

- Reading string from user:

```
User_name=raw_input(‘Enter your name:’)
```

- Reading result from command line

```
root=‘/home/doutriau/’
cmd=‘ls ‘+root
files=os.popen(cmd).readlines()
for file in files:
   file_name=file.strip() # removes trailing
  line carriage character
   print file_name
```

Files

- In Python, files are “objects”. To open an existing file:

  `f=open(‘file.txt’)`

- To open an existing file with possibility to write to it

  `f=open(‘file.txt’,’r+’)`

- To open a new file

  `f=open(‘file.txt’,’w’)`

- To read a file:

  `lines=f.readlines()` # lines is a list of string, each string ending up with the carriage return character (‘\n’)

- To write to a file:

  `f.writelines(lines)` # lines being a list a strings

  Or

  `f.write(‘hi\n’)`

  Or directly with the print statement

  `print >>f, ‘Hi’`

Types: Final note

- To determine the type of a python object:

```
print type([2,3,4]) # <type 'list'>
a= [2,3,4] ; if type(a) == type([]):
        print ‘It is  a list’
```

- You can also use the types module

```
import types
print dir(types.type)
if type(a) == types.ListType:
  print ‘it is a list’
if type(A) in [types.ListType, types.TupleType]:
  print ‘a is a list or  a tuple’
```

- Or use the “pure” object oriented method:

```
if isinstance(a,list):
  print ‘it is  a list’
if isinstance(a,(list,tuple)):
  print ‘a is a list or a tuple’
```

  - This allows to check for object of type you created (non standard python objects)

Conditions

- **Operators**

  `equal: == , not equal: != , greater: > , less < , greater_equal: >= , less_equal: <=`

  in: in (if a in [1,2,3,4]:)

- **Mixing**: `and, or, not`

- **Test and condition block are determined by a SEMI-COLON (:) and then every line having the same INDENTATION**

if/elif/else

```
if a>b:
 print ‘a greater than b’
elif a<b:
 print ‘b greater than a’
else:
 print ‘a and b are equal’
```

Loops: For 1

- **“for”** is used to **loop** through elements of a **list/tuple** and assign the values from the list to a variable that is used inside a **block** which is repeated until all the elements of the list have been used.

- Again a **semi-colon and an indented block** will determine the extent of the code to be executed:

```
for a in [‘a’,1,’2’,[4,5,6]]:
  print a
print ‘Done’
```

Loops: For 2

- To do the “traditional” loop over 10 numbers, create a list using the “range” command:

```
range(6) # returns [0,1,2,3,4,5]
range(2,6) # returns [2,3,4,5]
range(3,8,3) # returns [3,6]
range(8,3,-1) # returns [8,7,6,5,4]
range(8,3,-3) # returns [8,5]

for I in range(10):
    print I,’*’,5,’=‘,I*5
```

- As much as possible DO NOT nest loops, it is EXTREMELY inefficient.

Loops: While/Else

- **“while”** is used to **repeat** a block as long as a **condition** is fulfilled, again a **semi-colon and an indented block** will determine the extent of the executed block. The **else** statement is executed **after** the condition is not true anymore:

```
a=11
while a>0:
  a=a-1
  print a
print ‘Done’
```

- To the “traditional” loop over 10 numbers, initialize a “counter” to a value, and check for its value, keep increasing it every time:

```
c=0
while c<10:
 print c,’*’,5,’=‘,c*5
   c=c+1
else:  # executed once the loop is exited
   print ‘done with this’
print ‘All Done!’
```

Blocks: Try/Except/Else

- **“try/except”** is used to catch errors, allowing you to decide how to respond to different errors. Again a semi-colon and an indented block will determine the extent of the loop:

```
mylist=[1,2,3,-4,0]
b=5.
for c in mylist:
   try:
        b=a/c
   except:
        print ‘Impossible to divide by ‘,b
   else:
        print ‘No matter what, we come here’
```

Functions (1)

- **Functions** are “defined” using the “**def**” keywords followed with “**:**” and indentation will define the extent of the function:

```
>>> def my_print_add_func(a,b):
...     print a,’+’,b,’=‘,a+b
>>> print my_print_add_func(3,4)
```

- Arguments can be passed “out of order” using their **keyword** name:

```
>>> print my_print_add_func(b=4,a=3)
```

Functions (2)

- Arguments can be defined with default values, which allows user to not pass these:

```
>>> def my_print_add_func(a,b=4):
...     print a,’+’,b,’=‘,a+b
>>> my_print_add_func(3)
>>> my_print_add_func(3,6)
```

- Functions can return objects:

```
>>> def my_print_add_func(a,b):
...     return a+b
>>> c=my_print_add_func(a,b)
```

Functions (3)

- Unlimited number of arguments can be accepted via “*****” identifier:

```
>>> def my_add_func(*args):
    # args is a list of all args passed in
...   tot=0
...   for a in args:
...      tot=tot+a
...   return tot
>>> print my_add_func(1,2,3,4,5,6)
```

Functions (4)

- Unlimited number of “keyword arguments” can be accepted via the “**” identifier:

```
>>> def my_add_func(**kw):
...    # kw: dictionary containing pairs
...    # (keyword:value)
...    for v in kw.values():
...      tot=tot+v
...    return tot
>>> print my_add_func(a=5,b=6)
```

Classes

- Python allows Object-oriented programming. **Classes** define python objects, from which **instances** are created.
- Resulting objects have functions and attributes associated with them.
- Classes define the “blueprint” of these object and have an initialization function:

```
class TwoNumberOperators:
    def __init__(self,one=3,two=4):     # class methods are like functions
            # self refers to the object that will be created
            self.one=one
            self.two=two
    def add(self):
            return self.one+self.two
    def sub(self):
            return self.one-self.two
T1= TwoNumberOperators(one=6,two=5)
print T1.one,T1.two
print T1.add()
```

Modules

- **Modules** allows you to store function for use at a later time.
- Modules are named with the “.py” extension:

```
<yourfile.py>
```

- import other modules (or packages (i.e. groups of modules)) as follows:

```
import yourfile
from yourfile import yourFunction
```

- Access objects in the module namespace:

```
x=yourfile.yourFunction()
c=yourfile.myClass(some_argument)
s=yourfile.astring
```

Modules vs scripts

- In *pythonspeak*, there are two types of file:

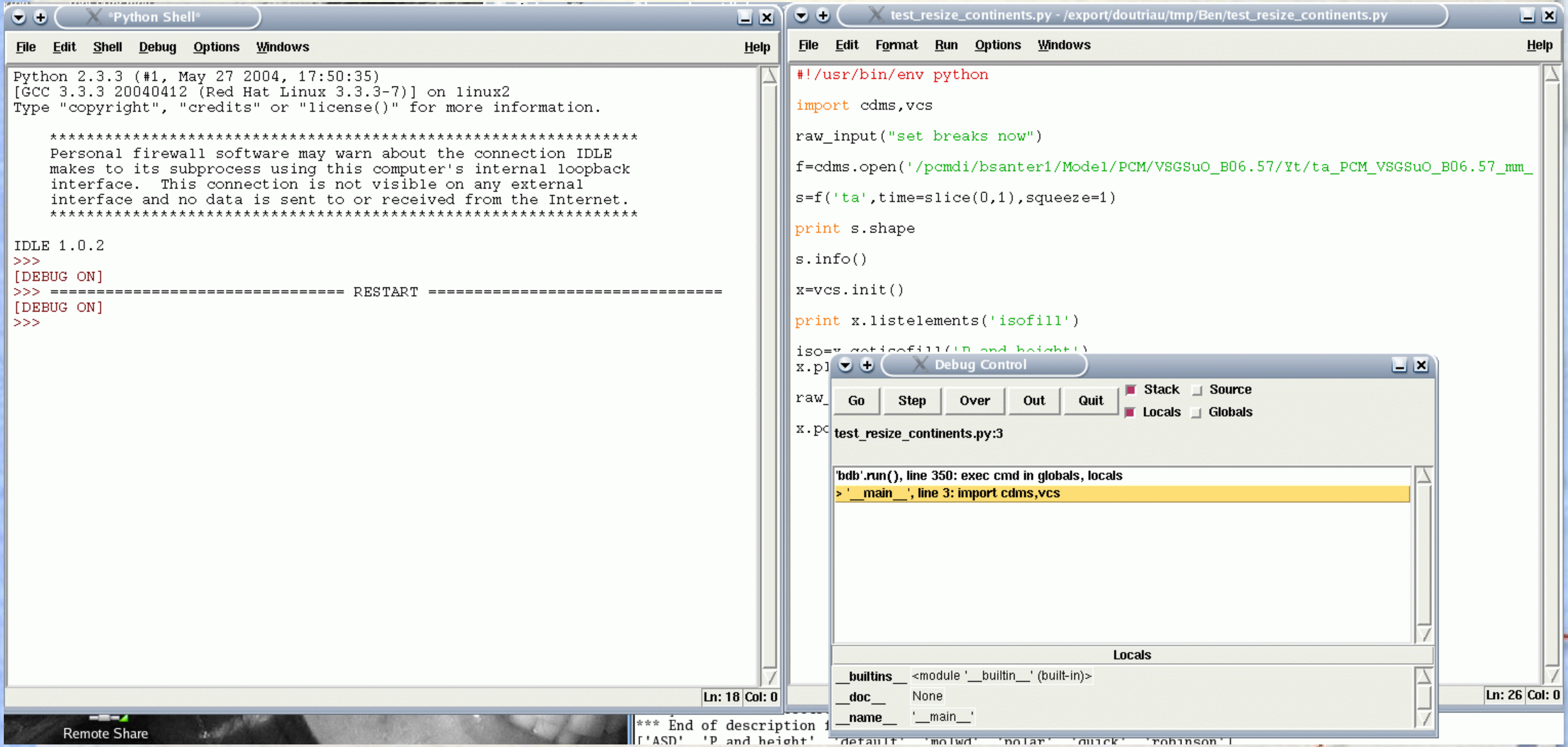
1. A **script** (or a *file*) contains statements and can be executed directly from the command-line (or from an icon in a Windows world).

2. A **module** should only contain definitions of functions, variables and and class constructors.

NOTE 1: A script can also contain definitions like a module.

Debugger - IDLE

- Python's editor, IDLE comes with a debugger
  - From the shell window check debugger and then run a loaded file (F5)



```
Python 2.3.3 (#1, May 27 2004, 17:50:35)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0.2
>>>
[DEBUG ON]
>>> ================================ RESTART ================================
[DEBUG ON]
>>>
```

```
#!/usr/bin/env python

import cdms,vcs

raw_input("set breaks now")

f=cdms.open('/pcmdi/bsanter1/Model/PCM/VSGSuO_B06.57/Yt/ta_PCM_VSGSuO_B06.57_mm_

s=f('ta',time=slice(0,1),squeeze=1)

print s.shape

s.info()

x=vcs.init()

print x.listelements('isofill')

iso=x.getisofill('P and height')
x.pl

raw_

x.pc
```

```
test_resize_continents.py:3

'bdb'.run(), line 350: exec cmd in globals, locals
> '__main__', line 3: import cdms,vcs
```

Locals

__builtins__	<module '__builtin__' (built-in)>
__doc__	None
__name__	'__main__'

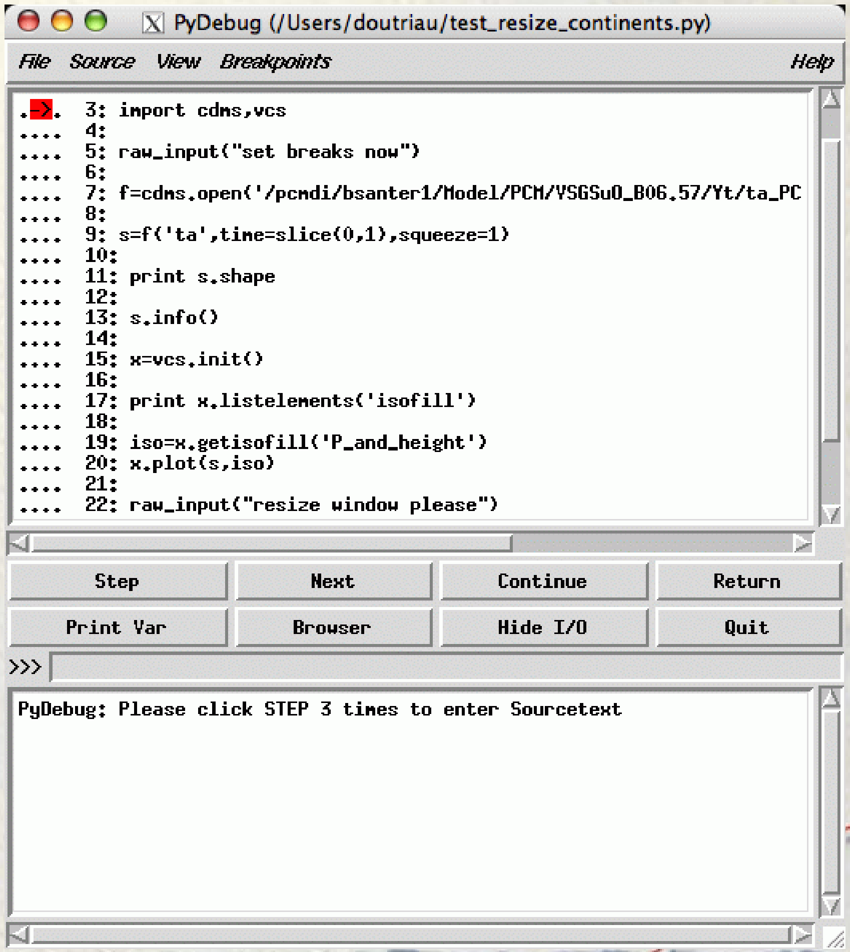
```
*** End of description f
['ASD', 'P and height', 'default', 'mollwd', 'polar', 'quick', 'robinson']
```

Python Shell
File Edit Shell Debug Options Windows Help
Ln: 18 Col: 0
test_resize_continents.py - /export/doutriau/tmp/Ben/test_resize_continents.py
File Edit Format Run Options Windows Help
Ln: 26 Col: 0
Debug Control
Go Step Over Out Quit
Stack Source
Locals Globals
Remote Share

Debugger - Pydebug

- CDAT comes with a debugger "pydebug"
  - It sits in your bin directory, and you can use it as follow:

```
pydebug python_file
```



```
.->.  3: import cdms,vcs
....  4:
....  5: raw_input("set breaks now")
....  6:
....  7: f=cdms.open('/pcmdi/bsanter1/Model/PCM/VSGSuO_B06.57/Yt/ta_PC
....  8:
....  9: s=f('ta',time=slice(0,1),squeeze=1)
.... 10:
.... 11: print s.shape
.... 12:
.... 13: s.info()
.... 14:
.... 15: x=vcs.init()
.... 16:
.... 17: print x.listelements('isofill')
.... 18:
.... 19: iso=x.getisofill('P_and_height')
.... 20: x.plot(s,iso)
.... 21:
.... 22: raw_input("resize window please")
```

```
PyDebug: Please click STEP 3 times to enter Sourcetext
```

Appendix for session: Introduction to python 1

Functions – built-ins

- There is a number of “built-in” functions in Python, a complete list is available at: http://www.python.org/doc/lib/built-in-funcs.html.
- Examples are:

  `help([`*`object`*`])` - Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated. New in version 2.2.

  `str([`*`object`*`])` - Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with repr(*object*) is that str(*object*) does not always attempt to return a string that is acceptable to eval(); its goal is to return a printable string. If no argument is given, returns the empty string, “”.

Functions – built-ins

- There is a number of “built-in” functions in Python, a complete list is available at: http://www.python.org/doc/lib/built-in-funcs.html

- **abs**(*x*) Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.
- **chr**(*i*) Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of `ord()`. The argument must be in the range [0..255], inclusive; `ValueError` will be raised if *i* is outside that range.
- **delattr**(*object, name*) This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(`*x*, `'`*foobar*`')` is equivalent to `del` *x.foobar*.

Functions – built-ins

- **dir**([*object*]) Without arguments, return the list of names in the current local symbol table. With an argument, attempts to return a list of valid attributes for that object. This information is gleaned from the object's `__dict__` attribute, if defined, and from the class or type object. The list is not necessarily complete. If the object is a module object, the list contains the names of the module's attributes. If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases. Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes. The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()
['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)
['__doc__', '__name__', 'calcsize', 'error', 'pack', 'unpack']
```

Note: Because `dir()` is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases.

Functions – built-ins

- **dir**([*object*]) Without arguments, return the list of names in the current local symbol table. With an argument, attempts to return a list of valid attributes for that object. This information is gleaned from the object's __dict__ attribute, if defined, and from the class or type object. The list is not necessarily complete. If the object is a module object, the list contains the names of the module's attributes. If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases. Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes. The resulting list is sorted alphabetically. For example:

```
>>> import struct>>> dir()['__builtins__', '__doc__', '__name__', 'struct']
>>> dir(struct)['__doc__', '__name__', 'calcsize', 'error', 'pack', 'unpack']
```

Note: Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases.

Functions – built-ins

- **divmod**(*a, b*) Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as (*a* / *b*, *a* % *b*). For floating point numbers the result is (*q*, *a* % *b*), where *q* is usually math.floor(*a* / *b*) but may be 1 less than that. In any case *q* * *b* + *a* % *b* is very close to *a*, if *a* % *b* is non-zero it has the same sign as *b*, and 0 <= abs(*a* % *b*) < abs(*b*). Changed in version 2.3: Using divmod() with complex numbers is deprecated.

Functions – built-ins

- **eval**(*expression*[, *globals*[, *locals*]]) The arguments are a string and two optional dictionaries. The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local name space. If the *globals* dictionary is present and lacks '__builtins__', the current globals are copied into *globals* before *expression* is parsed. This means that *expression* normally has full access to the standard __builtin__ module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where eval is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by compile()). In this case pass a code object instead of a string. The code object must have been compiled passing 'eval' as the *kind* argument. Hints: dynamic execution of statements is supported by the exec statement. Execution of statements from a file is supported by the execfile() function. The globals() and locals() functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by eval() or execfile().

Functions – built-ins

- **execfile**(*filename*[, *globals*[, *locals*]]) This function is similar to the exec statement, but parses a file instead of a string. It is different from the import statement in that it does not use the module administration -- it reads the file unconditionally and does not create a new module.2.2 The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local namespace. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where execfile() is called. The return value is None.

  Warning: The default *locals* act as described for function locals() below: modifications to the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function execfile() returns. execfile() cannot be used reliably to modify a function's locals.

Functions – built-ins

- **float**([*x*]) Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace; this behaves identical to string.atof(*x*). Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns 0.0. **Note:** When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. The specific set of strings accepted which cause these values to be returned depends entirely on the C library and is known to vary.
- **getattr**(*object, name*[*, default*]) Return the value of the named attributed of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, getattr(x, 'foobar') is equivalent to x.foobar. If the named attribute does not exist, *default* is returned if provided, otherwise AttributeError is raised.

Functions – built-ins

- **hasattr**(*object, name*) The arguments are an object and a string. The result is True if the string is the name of one of the object's attributes, False if not. (This is implemented by calling getattr(*object*, *name*) and seeing whether it raises an exception or not.)
- **help**([*object*]) Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated. New in version 2.2.

Functions – built-ins

- **input**([*prompt*]) Equivalent to eval(raw_input(*prompt*)). **Warning:** This function is not safe from user errors! It expects a valid Python expression as input; if the input is not syntactically valid, a SyntaxError will be raised. Other exceptions may be raised if there is an error during evaluation. (On the other hand, sometimes this is exactly what you need when writing a quick script for expert use.) If the readline module was loaded, then input() will use it to provide elaborate line editing and history features. Consider using the raw_input() function for general input from users. **int**([*x*[, *radix*]]) Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace. The *radix* parameter gives the base for the conversion and may be any integer in the range [2, 36], or zero. If *radix* is zero, the proper radix is guessed based on the contents of string; the interpretation is the same as for integer literals. If *radix* is specified and *x* is not a string, TypeError is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers truncates (towards zero). If the argument is outside the integer range a long object will be returned instead. If no arguments are given, returns 0.

Functions – built-ins

- **isinstance**(*object, classinfo*) Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct or indirect) subclass thereof. Also return true if *classinfo* is a type object and *object* is an object of that type. If *object* is not a class instance or an object of the given type, the function always returns false. If *classinfo* is neither a class object nor a type object, it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a TypeError exception is raised. Changed in version 2.2: Support for a tuple of type information was added.

Functions – built-ins

- **len**(*s*) Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).
- **list**([*sequence*]) Return a list whose items are the same and in the same order as *sequence*'s items. *sequence* may be either a sequence, a container that supports iteration, or an iterator object. If *sequence* is already a list, a copy is made and returned, similar to *sequence*[:]. For instance, list('abc') returns ['a', 'b', 'c'] and list((1, 2, 3)) returns [1, 2, 3]. If no argument is given, returns a new empty list, [].
- **long**([*x*[, *radix*]]) Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace; this behaves identical to string.atol(*x*). The *radix* argument is interpreted in the same way as for int(), and may only be given when *x* is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns 0L.

Functions – built-ins

- **map**(*function, list, ...*) Apply *function* to every item of *list* and return a list of the results. If additional *list* arguments are passed, *function* must take that many arguments and is applied to the items of all lists in parallel; if a list is shorter than another it is assumed to be extended with None items. If *function* is None, the identity function is assumed; if there are multiple list arguments, map() returns a list consisting of tuples containing the corresponding items from all lists (a kind of transpose operation). The *list* arguments may be any kind of sequence; the result is always a list.
- **max**(*s*[*, args...*]) With a single argument *s*, return the largest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the largest of the arguments.
- **min**(*s*[*, args...*]) With a single argument *s*, return the smallest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, return the smallest of the arguments.

Functions – built-ins

- **open/file**(*filename*[*, mode*[*, bufsize*]]) Return a new file object (described in section 2.3.8, ``File Objects''). The first two arguments are the same as for stdio's fopen(): *filename* is the file name to be opened, *mode* indicates how the file is to be opened: 'r' for reading, 'w' for writing (truncating an existing file), and 'a' opens it for appending (which on *some* Unix systems means that *all* writes append to the end of the file, regardless of the current seek position). Modes 'r+', 'w+' and 'a+' open the file for updating (note that 'w+' truncates the file). Append 'b' to the mode to open the file in binary mode, on systems that differentiate between binary and text files (else it is ignored). If the file cannot be opened, IOError is raised. In addition to the standard fopen() values *mode* may be 'U' or 'rU'. If Python is built with universal newline support (the default) the file is opened as a text file, but lines may be terminated by any of '¥n', the Unix end-of-line convention, '¥r', the Macintosh convention or '¥r¥n', the Windows convention.

Functions – built-ins

- **open/file** (continued). All of these external representations are seen as '¥n' by the Python program. If Python is built without universal newline support *mode* 'U' is the same as normal text mode. Note that file objects so opened also have an attribute called newlines which has a value of None (if no newlines have yet been seen), '¥n', '¥r', '¥r¥n', or a tuple containing all the newline types seen. If *mode* is omitted, it defaults to 'r'. When opening a binary file, you should append 'b' to the *mode* value for improved portability. (It's useful even on systems which don't treat binary and text files differently, where it serves as documentation.) The optional *bufsize* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used.2.3 The file() constructor is new in Python 2.2. The previous spelling, open(), is retained for compatibility, and is an alias for file().

Functions – built-ins

- **pow**(*x, y*[*, z*]) Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than pow(*x*, *y*) % *z*). The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, 10**2 returns 100, but 10**-2 returns 0.01. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument pow() returned platform-dependent results depending on floating-point rounding accidents.)

Functions – built-ins

- **range**([*start,*] *stop*[*, step*]) This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. The full form returns a list of plain integers [*start*, *start* + *step*, *start* + 2 * *step*, ...]. If *step* is positive, the last element is the largest *start* + *i* * *step* less than *stop*; if *step* is negative, the last element is the largest *start* + *i* * *step* greater than *stop*. *step* must not be zero (or else ValueError is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0)
[]
```

Functions – built-ins

- **raw_input**([*prompt*]) If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised. Example:

  ```
  >>> s = raw_input('--> ')
  --> Monty Python's Flying Circus
  >>> s
  "Monty Python's Flying Circus"
  ```

  If the readline module was loaded, then raw_input() will use it to provide elaborate line editing and history features.

- **reduce**(*function, sequence*[*, initializer*]) Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

Functions – built-ins

- **reload**(*module*) Re-parse and re-initialize an already imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument). There are a number of caveats: If a module is syntactically correct but its initialization fails, the first import statement for it does not bind its name locally, but does store a (partially initialized) module object in sys.modules. To reload the module you must first import it again (this will bind the name to the partially initialized module object) before you can reload() it. When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains.

Functions – built-ins

- **reload** (continued). This feature can be used to the module's advantage if it maintains a global table or cache of objects -- with a try statement it can test for the table's presence and skip its initialization if desired. It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for sys, __main__ and __builtin__. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded. If a module imports objects from another module using from ... import ..., calling reload() for the other module does not redefine the objects imported from it -- one way around this is to re-execute the from statement, another is to use import and qualified names (*module*.*name*) instead. If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances -- they continue to use the old class definition. The same is true for derived classes.

Functions – built-ins

- **repr**(*object*) Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval().
- **round**(*x*[*, n*]) Return the floating point value *x* rounded to *n* digits after the decimal point. If *n* is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus *n*; if two multiples are equally close, rounding is done away from 0 (so. for example, round(0.5) is 1.0 and round(-0.5) is -1.0).

Functions – built-ins

- **setattr**(*object, name, value*) This is the counterpart of getattr(). The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, setattr(*x*, '*foobar*', 123) is equivalent to *x*.*foobar* = 123.
- **slice**([*start,*] *stop*[*, step*]) Return a slice object representing the set of indices specified by range(*start*, *stop*, *step*). The *start* and *step* arguments default to None. Slice objects have read-only data attributes start, stop and step which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: "a[start:stop:step]" or "a[start:stop, i]".
- **str**([*object*]) Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with repr(*object*) is that str(*object*) does not always attempt to return a string that is acceptable to eval(); its goal is to return a printable string. If no argument is given, returns the empty string, ''.

Functions – built-ins

- **sum**(*sequence*[, *start*]) Sums *start* and the items of a *sequence*, from left to right, and returns the total. *start* defaults to 0. The *sequence*'s items are normally numbers, and are not allowed to be strings. The fast, correct way to concatenate sequence of strings is by calling ''.join(*sequence*). Note that sum(range(*n*), *m*) is equivalent to reduce(operator.add, range(*n*), *m*) New in version 2.3.

Functions – built-ins

- **tuple**([*sequence*]) Return a tuple whose items are the same and in the same order as *sequence*'s items. *sequence* may be a sequence, a container that supports iteration, or an iterator object. If *sequence* is already a tuple, it is returned unchanged. For instance, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, returns a new empty tuple, ().
- **type**(*object*) Return the type of an *object*. The return value is a type object. The standard module types defines names for all built-in types that don't already have built-in names. For instance

```
:>>> import types
>>> x = 'abc
'>>> if type(x) is str: print "It's a string”
...
It's a string
>>> def f(): pass
...
>>> if type(f) is types.FunctionType: print "It's a function”
...
It's a function
```

The isinstance() built-in function is recommended for testing the type of an object.